

A Generative Model of Software Dependency Graphs to Better Understand Software Evolution

Vincenzo Musco^{*}, Martin Monperrus[†], Philippe Preux[‡]
University of Lille, CRIStAL, France

Email: ^{*}vincenzo.musco@inria.fr, [†]martin.monperrus@univ-lille1.fr, [‡]philippe.preux@univ-lille3.fr

Abstract—Software systems are composed of many interacting elements. A natural way to abstract over software systems is to model them as graphs. In this paper we consider software dependency graphs of object-oriented software written in the Java programming language and we study one topological property: the degree distribution. Based on the analysis of fifteen software systems written in Java, we show that there exists completely different systems that have the same degree distribution. This motivates us to propose a generative model of software dependency graphs which synthesizes graphs whose degree distribution is close to the empirical ones observed in real software systems. This model gives us novel insights on the potential hidden rules of software evolution.

Index Terms—Software Engineering, Dependency Graphs, Degree Distribution, Software Evolution.



1 INTRODUCTION

Software systems are composed of many elements interacting with each other. For instance there are hundreds of thousands of interconnected functions in a Linux kernel [1]. A natural way to abstract over software systems is to model them as graphs [2], [3], [4], [5], [6].

In this paper we consider software dependency graphs of object-oriented software, where each node represents a class and each edge corresponds to a compile-time dependency. We study the structure of those graphs and we specifically concentrate on their degree distributions, that is the distribution of the number of edges coming from and to a node. Based on the analysis of fifteen software systems written in Java, we show that there exists different software systems that have similar degree distribution.

This is a surprising result: despite being developed by different persons with different processes in different domains, a common degree distribution emerges. This makes us hypothesize that whatever the development design, the structure of software dependency graphs tend to be similar.

In network science, a generative model defines a set of rules used to synthesize artificial graphs¹ in a given domain. For example, there exists generative models that aim at producing graphs that are similar to the World Wide Web [7]. One equivalent model in software engineering would be a model generating graphs which look like real software graphs. Having a good generative model of software graphs would allow answering hard yet essential research questions: what are the software evolution rules driving the construction of graph structures of software systems? how does a given software graph at a point in time will evolve in the future?

In this paper, we concentrate on the first question. Our goal is to propose a generative model of software dependency graphs. If this model produces graphs that fit the empirical data, it would mean that the generative operations are good candidates to describe the core operations which result in software graphs' structure. In other words, a good generative model may encode the evolution rules that are behind the graph structure of software systems.

Our experimental methodology is as follows. We propose a generative model of software graphs, and then we evaluate its capacity to create graphs whose degree distribution is close to the empirical ones observed in real software systems. We compare its fit-to-data to the only comparable model of the literature [2]. Our experimental results show that our generative model of software dependency graphs, GD-GNC, both fits the empirical data and outperforms the model proposed in [2]. To put it shortly, our generative model is the first to produce software dependency graphs that look like real ones.

To sum up, our contributions are:

- empirical evidence of the common asymmetric structure of dependency graphs in object-oriented software systems written in Java: the in-degree distribution is different from the out-degree distribution;
- a generative model of software dependency graphs;
- the validation of the model regarding its ability to fit fifteen graphs of real software systems totaling 23178 nodes and 108404 edges;
- a speculative explanation of the evolution rules of software.

The rest of this paper is structured as follows. Section 2 defines the main concepts used in this paper. In Sec-

1. In this paper, we use “network” and “graph” interchangeably.

tion 3, we look closer at real software dependency graphs and highlight their common structure. In Section 4, we introduce a new generative model for software dependency graphs and we analyze its fitness. In Section 5, we discuss our findings from a software engineering perspective.

2 BACKGROUND

In this section, we provide background knowledge about the concepts used in this paper.

2.1 Graphs, Degrees and Distributions

Let $G(N, E)$ be a directed graph composed of N nodes and E directed edges connecting nodes to each other. An edge e connecting node n_1 to node n_2 is expressed as an ordered pair: $e = (n_i, n_j)$, where $e \in E, n_i, n_j \in N$.

The *in-degree* and the *out-degree* of a node n_i are respectively the number of edges going to n_j (i.e. the number of edges (\cdot, n_j)) and the number of edges leaving n_i (i.e. the number of edges (n_i, \cdot)). We use the term *degree* to refer to both of those concepts, and terms *in-degree* and *out-degree* when the distinction is necessary.

The *degree distribution* of a graph is the proportion of each degree in that graph. It sums to 1. In this paper, we always consider *cumulative distribution functions* (CDF) of degrees, i.e. the proportion of nodes whose degree is smaller or equal to a given value. The main reason is that noncumulative distributions are to be avoided as they are sources of mistakes [8]. Cumulative distributions are more appropriate to analyze noisy and right-skewed distributions [9], which is the type of distributions we have. Finally, the related literature also considers *inverse cumulative distributions* (ICD), we will do the same for the sake of easy comparison.

Node degrees and their distributions are basic properties which are directly influenced by many properties of a graph. For instance, the edge density and the clustering coefficient are directly correlated to it. The *graph structure* refers to the type and abundance of edge patterns [10, sec. 8.3]. Beyond this abstract definition, graph metrics focus on different aspects of the graph structure: the graph diameter, the degree distributions, etc. As others, the key assumption of our work is the cumulative degree distribution reflects an essential aspect of the graph structure [2], [3], [5], [6], [11], [12], [13].

2.2 Software Dependency Graphs

A large variety of graphs can be derived from software, each one focusing on particular characteristics. Hence, nodes and edges can have various meanings. An example of software graph is the *dependency graph* in which nodes are modules (e.g. packages, classes, etc. depending on the chosen granularity) and edges reflect that an element uses another one (e.g. function call, inheritance, field access, etc.).

Dependency graphs are directed graphs as dependencies are oriented. The nodes composing a dependency graph can be of two different types. First, there are *application nodes* (a.k.a. app nodes) that are nodes which belong to the core software itself. Second, there are *library nodes* (a.k.a. lib nodes) which are nodes which belong to an external library. For instance, a class of software package “Eclipse” may use a class in `java.util` library. The former is an application node, the latter is a library node.

From this distinction between the two types of nodes, there are two types of edges in a software dependency graph: app-app edges application nodes to application nodes, we call them *endo-dependencies*, they express that a core class depends on another core class; app-lib edges application nodes to library nodes, we call them *exo-dependencies*, they express that a core class depends on a library class. Fig. 1 illustrates these notions. On the left-hand side on the figure are emphasized endo-dependencies, on the right-hand side, the bold red arrows crossing the system boundary are exo-dependencies. In this paper, we exclusively focus on endo-dependencies.

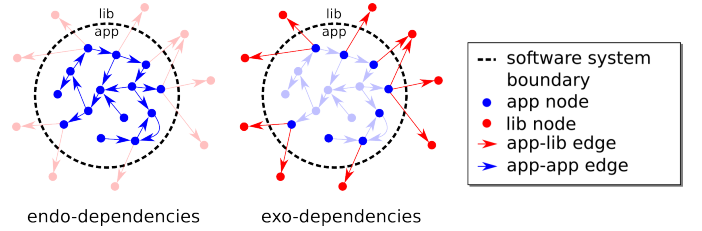


Fig. 1: Illustration of the set of endo-dependencies (left) and exo-dependencies (right) in a dependency graph.

2.3 Generative Models

A generative model for graphs is an algorithm that generates graphs. A generative model takes a set of parameters as input (such as the number of nodes and parameters that influence how nodes connect to each other). A generative model may be *deterministic* or *stochastic*. For a given set of parameters, a deterministic model always generate the exact same graph whereas a stochastic model generates a new graph each time it is run.

In our case, we generate graphs that are intended to look like software dependency graphs. We consider two types of graphs: those resulting from an analysis of software systems, and those created by a generative model. The former are qualified as “empirical” or “true”, the latter being qualified as “synthetic” or “artificial”.

3 STUDY OF THE COMMON STRUCTURE OF SOFTWARE DEPENDENCY GRAPHS

We want to determine whether there exists structures shared by different software applications. As the production of those software packages is influenced by different

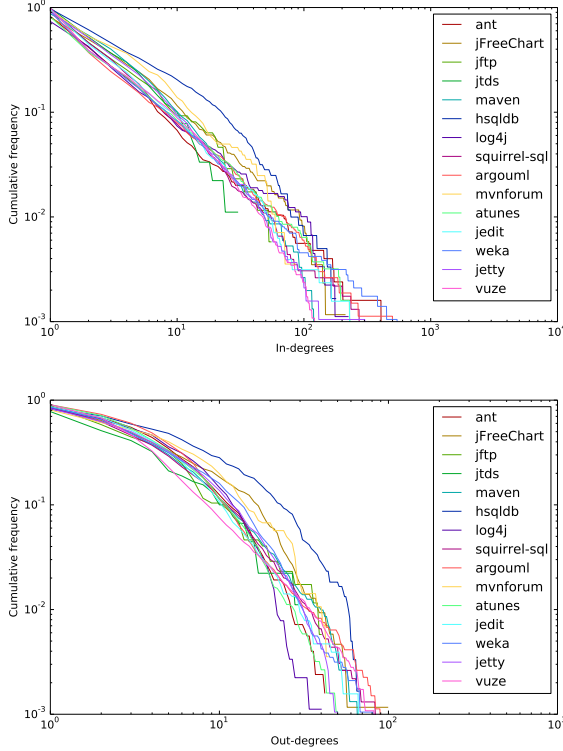


Fig. 2: Inverse cumulative in- and out-degree distribution for the 15 software applications of our dataset (axes on a logarithmic scale). It is clear from these plots that the in-degree ICD and the out-degree ICD are different.

factors (management and development teams, development techniques, ...), it is *a priori* expected that there is no common structure. On the opposite, finding common structures would be an interesting fact, it would mean that there exists common evolution mechanisms across application domains and development styles. Hence, our first research question reads:

Research Question 1 Are there common structures of software dependency graphs?

3.1 Protocol

Our protocol consists of a technique to extract graphs from software code, applied to a given dataset, and a statistically sound analysis method.

3.1.1 Dependency Graph Extraction

We now present our method to extract dependency graphs. We consider object-oriented software written in Java. We focus on the *class granularity* (i.e. one node represents one class), as this is the most important modularity unit in object-oriented software. We only consider endo-dependencies, that is, edges connecting internal nodes of the project to each others and not those connecting to external libraries (cf. section 2.2), because we aim at understanding the inner structure of software graphs, regardless of the number of libraries that are included and the amount of calls to library functions.

This extraction is performed by Dependency Finder². This mature and open-source tool takes as input Java byte code and outputs all dependencies being found. Graph metrics are computed using the NetworkX³ library.

3.1.2 Dataset

To determine an acceptable dataset size, we examined the dataset size in related publications: they range from 1 to 12 [2], [3], [5], [6], [11], [12], [13]. Consequently, we build a dataset of 15 software applications. We ensure that they are all developed by different teams, in order to reduce the risk of having results biased towards specific development processes. Since our graph extraction tool chain handles Java software, we consider software that is open-source and written in Java. Note that Louridas et al. [5] have shown that the graph structure is independent of the programming language. Finally, we select software applications which are different in size (number of nodes/edges).

The resulting dataset⁴ is presented in Table 1. For each software, this table contains its name, the version that has been used in this study, the number of nodes $|N|$ and the number of edges $|E|$ contained in the graphs we extract, and the graph density γ . The graph density is computed using formula (1). In words, the graph density expresses the proportion of node pairs being connected in the graph.

$$\gamma = \frac{|E|}{|N| \times (|N| - 1)} \quad (1)$$

The graph size ranges from 90 to 4633 nodes, from 328 to 18493 edges and has a graph density γ value ranging from 0.001 to 0.04.

3.1.3 Comparison of inverse cumulative distributions

In this paper, our main analysis tool is the comparison of degree CDFs. To compare two CDFs, we use the Kolmogorov-Smirnov statistic K (also called distance) which measures a distance between two distributions. The statistic is given in formula (2) where \sup is the supremum of a set, F_1 and F_2 are the two distributions to compare and x ranges over degree values.

$$K_{F_1, F_2} = \sup_x |F_1(x) - F_2(x)| \quad (2)$$

K is a numerical value that indicates how close two distributions are: the lower K , the closer the distributions.

For one experiment presented in this paper, we perform a Mann-Whitney U test [14] on K to compare two generative models.

2. <http://depfind.sourceforge.net/>

3. <http://networkx.lanl.gov/>

4. The dataset can be downloaded at http://www.vmusco.com/pages/dataset_generative_model.html

TABLE 1: The 15 Java programs considered in our experiments: we provide their names, the version that has been used, and basic statistics of their dependency graphs: the number of nodes $|N|$, the number of edges $|E|$, and the density γ .

	Version	$ N $	$ E $	γ
ant	1.9.2	1252	5763	0.004
jfreechart	1.0.16	858	4783	0.006
jftp	1.57	173	736	0.025
jtds	1.3.1	90	328	0.040
maven	3.3.1	1515	6933	0.003
hsqldb	2.3.1	602	4976	0.014
log4j	2.0b9	895	4136	0.005
squirreelsql	3.5.0	2288	10141	0.002
argouml	0.34	2664	13445	0.002
mvnforum	1.3	282	1614	0.020
atunes	3.1.2	1881	8502	0.002
jedit	5.2.0	1277	5674	0.003
weka	3.7.12	2860	14082	0.002
jetty	9.2.7	1908	8798	0.002
vuze	5.5	4633	18493	0.001
Total		23178	108404	

3.2 Results

Figure 2 shows the plot of the in-degree ICD (top) and out-degree ICD (bottom) for our dataset. The scale is bi-logarithmic. There is a different line and color for each software of the dataset. Recall that the ICD against degrees is the number of nodes in the graph which have a degree greater than or equal to a certain x .

We observe that: (i) in-degree and out-degree ICD are not the same: in-degree ICD is a straight line in log-log scale whereas out-degree ICD is more curved. (ii) for in-degree, the position and the shape of ICDs are graphically similar, this indicates there are common structures across software applications. This point also holds for the out-degree distribution.

Observation (i) has already been made in previous works [6], [12]. There is an asymmetry of in-degree and out-degree distributions [15]. We also note that our empirical data suggests power law distributions. This has already been much studied [5], [11]; however, this point is controversial [16] and is out of the scope of this paper.

In this paper, we focus on observation (i) for two reasons. First, this point has not been deeply studied so far; in particular, it has not been studied with a strong statistical assessment. Second, it is a required step before building a generic generative model for software graphs.

3.3 Statistical significance

In order to assess observation (i) in a statistical manner, we now express our null hypothesis and the alternate hypothesis:

TABLE 2: Number of times the H_0 hypothesis is rejected or accepted for in-degree, out-degree and both CDFs according to the two-sample Kolmogorov-Smirnov test with a confidence level α of 0.01.

	H_0 Rejected		H_0 Not rejected	
	Count	Ratio	Count	Ratio
In	48/105	46%	57/105	54%
Out	33/105	31%	72/105	69%
Total	81/210	39%	129/210	61%

H_0 : Samples from the software in-degree distributions (resp. out-degree) are drawn from the same distribution.

H_1 : Samples from the software in-degree distributions (resp. out-degree) are not drawn from the same distribution.

Using the *two-sample Kolmogorov-Smirnov test* on each pair of degree distributions of the dataset, we can statistically determine the presence of a similar structure across software applications in our dataset. Based on the statistic K , one decides on rejecting H_0 or not according to a given confidence level. If H_0 is not rejected, we gain confidence about the common structure for those two software. In the other hand, if H_0 is rejected, the test outcome can not be used to conclude about the common structure (which does not necessarily means that the two software graphs are not drawn from the same distribution).

Table 2 gives results for running 210 two-sample Kolmogorov-Smirnov tests with a confidence level α of 0.01⁵. The rows provide the results for in-degree, out-degree, and both distributions. The second and third columns provide the number and the ratio of tests for which the two-sample Kolmogorov-Smirnov test has rejected H_0 . The third and the fourth columns provide the opposite data (*i.e.* the test has not rejected H_0).

As we can see, for 61% of the tested pairs, the common distribution hypothesis cannot be rejected. However, this affirmation does not necessarily involve that there is a unique distribution shared by all those software. On the other hand, for the remaining 39% of tested pairs for which H_0 is rejected at this confidence level, no conclusion can be drawn.

3.4 Summary

To sum up, we reply positively to our first research question: our experiment indicates that, according to the degree distribution, there are common structures across software dependency graphs. This is not true for any arbitrary pair of software applications, but this holds for the majority. *Hence, we hypothesize that there is a common evolution process that eventually yields those common degree distributions.*

5. We need to test each pair of software, hence $C_{15}^2 = 105$ tests, which is doubled since we test in-degrees and out-degrees.

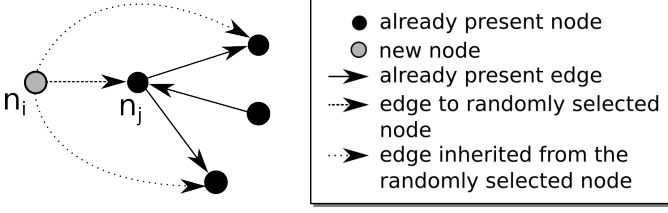


Fig. 3: Illustration of GNC-Attach, the GNC primitive operation. The grey node n_i is a new node added to the graph using the GNC primitive. The central node n_j is selected uniformly at random and a directed edge is added from the new node to it (dashed edge). Then, a directed edge is also added from the added node to all successors of n_j (dotted edges).

4 A GENERATIVE MODEL FOR SOFTWARE DEPENDENCY GRAPHS

In this section, we present a new generative model of software dependency graphs. This model generates an arbitrary number of artificial dependency graphs. It is parametrized by three values: the expected number of nodes and two probabilistics.

4.1 Generative Models of the Literature

To our knowledge, there exists three notable generative models explicitly targeting software graphs: one by Valverde and Sole [17], one by Myers [12] and one by Baxter and Freen [2]. According to our experiments, Baxter and Freen’s model [2] is the only one which gives reasonable degree distributions. The other ones are discarded because the resulting degree distributions are really different from the ones we observe in our dataset. Hence, we consider Baxter and Freen’s model as a baseline. Baxter and Freen’s model [2] encodes preferential attachment based on the out-degree of nodes. Its logic is based on 1) edge creation and 2) edge transfer between nodes of the graph: a transfer means that either the source or the destination of an edge is modified and points to another node of the graph.

There are also generic models, such as the one proposed in 1959 by Erdős & Rényi [18]. This model connects any pair of nodes according to a fixed probability p . The density of the resulting graph is hence $\gamma = p$. We will also use this model as a point of comparison.

4.2 Generalized Double GNC (GD-GNC)

We now present our generative model of software dependency graphs, called “Generalized Double GNC” (GD-GNC for short).

It is a generalization of the GNC model [13]: this model is an iterative algorithm where, at each iteration, a new node n_i is added to the graph and connected at random to a set of already existing nodes. To be more precise, an existing node n_j is selected according to a uniform distribution and directed edges are created

Algorithm 1: GNC-Attach Algorithm.

Input: $G_{\mathcal{N},\mathcal{E}}$ the digraph to which a new node n_i is going to be added. $G_{\mathcal{N},\mathcal{E}}$ is composed of two elements: the set of existing nodes (\mathcal{N}), and the set of existing directed edges (\mathcal{E}).

Output: $G_{\mathcal{N},\mathcal{E}}$ has been updated with a new node, and a set of new directed edges.

Function $\text{GNC_Attach}(G_{\mathcal{N},\mathcal{E}}, n_i)$ **is**
 Selects uniformly at random a node n_j in $G_{\mathcal{N},\mathcal{E}}$
 Add an edge from n_i to n_j
for all edges (n_j, n_d) leaving n_j **do**
 Add an edge from n_i to n_d
end for

from the new node n_i to this node n_j along with all its successors. This “GNC-Attach” primitive is illustrated in Figure 3. Algorithm 1 shows the core primitive for attaching nodes using GNC. GNC requires one parameter: the number of nodes of the resulting graph. It executes n times the core function to create a graph with n nodes.

GD-GNC consists in a main loop which at each iteration:

- adds a new node n_i to the existing graph,
- selects uniformly at random an existing node n_j ,
- adds edges leaving n_i .

The last step is performed as follows:

- with probability p , n_i is connected to n_j in the same way as in the GNC-Attach algorithm (*i.e.* a directed edge is created from n_i to n_j and from n_i to each successor of node n_j).
 Then, with probability q , we repeat this GNC-attachment once: an existing node is selected uniformly at random; if this selects the same node as previously, this second operation aborts.
- with probability $1 - p$, n_j is connected to n_i .

A pseudo-code is shown in Algorithm 2. GD-GNC is a generalization of GNC-Attach: GNC-Attach is a special case where $p = 1$ and $q = 0$.

We note that this model never modifies existing edges: at each iteration, it adds a single node and a set of edges. We emphasize the fact that no explicit preferential attachment is explicitly coded in the algorithm. However, an implicit preferential attachment is still present. GNC-Attach connects a new node to an existing one, but also to all successors of this existing node. As a consequence, if a node has a high in-degree value, it has a higher probability of receiving a new edge.

Two parameters are required by GD-GNC and influence the growth of the graph. p determines whether the new node n_i must be added following GNC-Attach, or whether it is connected to by an existing node. Hence, there is a proportion of p nodes that have no outgoing edges (sink nodes). q determines whether GNC-Attach should be executed once or twice for the inserting node. Increasing the number of GNC-Attach impacts the in-degree distribution, the distribution decreases more

Algorithm 2: Iterative algorithm for the “Generalized Double GNC” generative model.

Input: N the number of iterations to execute, p the probability to perform a GNC-Attach, and q the probability to do a double GNC-Attach.

Output: a digraph $G_{\mathcal{N},\mathcal{E}}$ is composed of two sets: nodes (\mathcal{N}) and directed edges (\mathcal{E})

```

begin
   $\mathcal{N} \leftarrow \emptyset$ 
   $\mathcal{E} \leftarrow \emptyset$ 
  for  $i \in \{1, \dots, N\}$  do
    Create a node  $n_i$  and add it to  $\mathcal{N}$ 
    if  $\text{rand}() \leq p$  then
      GNC_Attach( $G_{\mathcal{N},\mathcal{E}}, n_i$ )
      if  $\text{rand}() \leq q$  then
        GNC_Attach( $G_{\mathcal{N},\mathcal{E}}, n_i$ )
    else
      Selects uniformly at random a node  $n_j$  in  $G_{\mathcal{N},\mathcal{E}}$ 
      Add an edge from  $n_j$  to  $n_i$ 

```

slowly when q increases. Regarding the out-degree, the convexity of the distribution increases as q increases.

4.3 Evaluation of GD-GNC

We now want to determine whether the Generalized Double GNC can generate graphs that are more realistic than the graphs generated with Baxter & Fren’s model. We formulate this research question as:

Research Question 2 Do class dependency graphs generated using GD-GNC better fit real software data than Baxter & Fren’s model (according to the cumulative degree distribution)?

4.3.1 Protocol

To address this question, we first run a parameter optimization (see below) for each model (GD-GNC and Baxter & Fren) on all programs of our dataset (see Table 1). Then, we generate 30 synthetic graphs with each model, using the best parameters found for fitting each program. Finally, we compute the inverse cumulative degree distribution of each graph and we calculate the fitness value according to its δ value defined by Equation (3).

To evaluate whether a generative model is good, we use the Kolmogorov-Smirnov statistic [14] to compare artificial degree distributions with real ones. To determine the best values of p and q to generate graphs as close as possible to real ones, we perform a grid-search of the space of parameters. For each parameter value, we generate 30 graphs. Then, we use the K statistic to assess the fitness between the true graph and the generated one. The graph with the smallest K value is the one that looks like real data the most.

4.3.2 Results

For each software in our dataset, Figure 4 displays the in-degree ICD (top) and the out-degree ICD (bottom) of the real graph and that of artificial graphs generated using

various models. Each small plot represents a different software in our dataset. The meaning of the axis is the same for all such plots: x-axes are degrees and y-axes are degree ICDs. Both axis are in logarithmic scale. The thick continuous line corresponds to the real graph, the thin continuous line is for graphs generated using the GD-GNC model, and the dotted line is for graphs generated using Baxter & Fren’s model.

Graphically, we observe that GD-GNC in-degree and out-degree distributions are almost always better than Baxter & Fren’s. In other words, the GD-GNC algorithm produces generally synthetic software graphs whose inverse cumulative in-degree and out-degree distribution better fit those of real software dependency graphs than Baxter & Fren. We now move to a more serious assessment of the fit.

4.3.3 Statistical significance

To determine statistically which model generates the closest graph to the true one, we compare the Kolmogorov-Smirnov statistic K (as presented on section 3.1.3) for in-degree and out-degree cumulative degree distribution of the generated graph \mathcal{G} and the real graph \mathcal{R} . For this purpose, we define the δ function, as shown in Equation (3), which is the maximum between the two Kolmogorov-Smirnov distances: the distance $K_{\mathcal{R}_{in}, \mathcal{G}_{in}}$ between the in-degree cumulative distribution of the artificial graph \mathcal{G}_{in} and the real one \mathcal{R}_{in} , and likewise for the out-degree distribution (resp. $K_{\mathcal{R}_{in}, \mathcal{G}_{out}}$, \mathcal{G}_{out} , \mathcal{R}_{out}).

$$\delta_{\mathcal{R}, \mathcal{G}} = \max(K_{\mathcal{R}_{in}, \mathcal{G}_{in}}, K_{\mathcal{R}_{out}, \mathcal{G}_{out}}) \quad (3)$$

Combining in-degree and out-degree distributions is necessary because both distribution are intimately related to each other: considering only in-degree or out-degree distribution would be meaningless as a good in-distribution does not necessarily involve a good out-distribution and vice-versa.

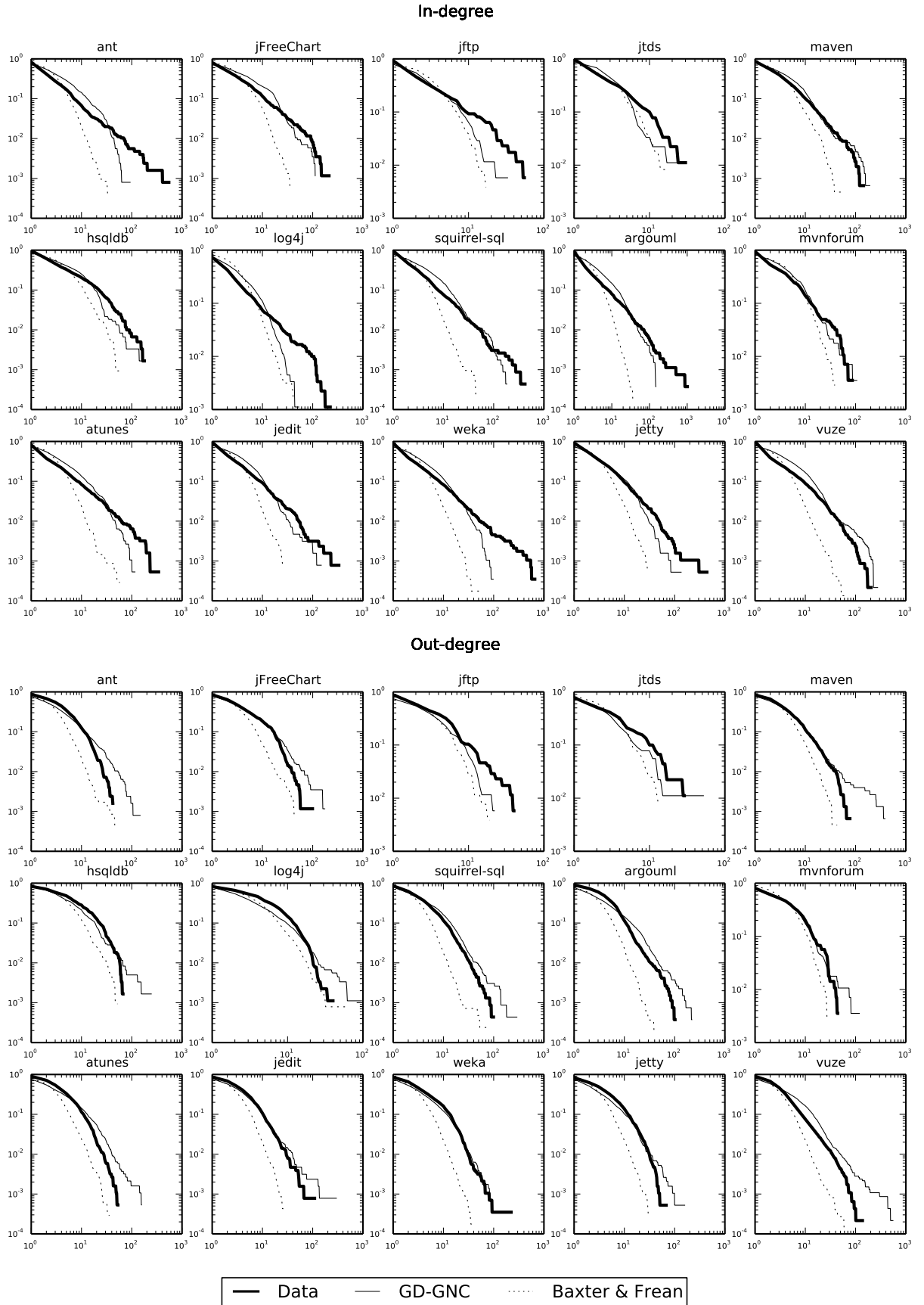


Fig. 4: Plot of the cumulative in- and out-degree distribution for 1) the real software graph (thick solid line); 2) the best generated match using the Generalized Double GNC model (thin solid line) 3) the best match generated using Baxter & Frean’s model (dotted line). Generalized Double GNC visually outperforms Baxter & Frean’s model.

As we want graphs which are similar to real ones according to their degree distribution and as the δ value represents the largest distance between a pair of distributions, considering in-degree and out-degree distributions, we know then the model which produces the smallest δ value is the best. To statistically ensure each δ value obtained for a model is drawn from a different distribution, we use the Mann-Whitney U test [14]. In terms of null hypothesis, this test allows us to reject or not the null hypothesis:

H_0 : the δ values obtained for GD-GNC and the ones obtained from Baxter & Frenan model belong to an identical population.

H_1 : the δ values obtained for GD-GNC and the ones obtained from Baxter & Frenan model belong to a different population.

Table 3 sums up values for δ obtained from 30 generated graphs with each model for each software. There is one row for each software of our dataset. In each row, columns report the name of the software, the minimal, median, and maximal δ values for Erdős & Rényi model, then for Baxter & Frenan and finally for GD-GNC. The last column is the p-value obtained using the Mann-Whitney test between GD-GNC and Baxter & Frenan.

Comparing graphs generated by Erdős & Rényi's model (columns 2–4) to GD-GNC (column 8–10) on the one hand, and Baxter & Frenan's (column 5–7) on the other hand, it is clear that both Baxter & Frenan's and GD-GNC models generate graphs more similar to real ones.

Furthermore, graphs generated using GD-GNC (column 8–10) are almost always more similar to real graph than those generated using the Baxter & Frenan's (column 5–7). However, for some software (maven, hsqldb and log4j if only the median value is considered), Baxter & Frenan's seems to generate better graphs. The Mann-Whitney p-value test shows those results are statistically significant as their p-value is lower than 0.05, excepted for log4j.

To sum up, according to our experiments on the degree distributions, our GD-GNC model is able to reproduce software structures more accurately than the Baxter & Frenan model.

5 DISCUSSION

We now put aside technical considerations and discuss the meaning and validity of our empirical results.

5.1 GD-GNC from a Software Engineering Perspective

We have a generative model which fits degree distributions of empirical software graphs. This model is only expressed in terms of primitive graph operations on nodes and edges, without any specific rule or knowledge from software engineering. Our initial intuition is that such a model implicitly captures certain software evolution rules. We now try to express these rules. In other words,

we now speculatively explain the model from a software engineering perspective.

The GD-GNC model is made-up of two basic operations (the top-level if/then/else of Algorithm 2).

The first basic operation of the model is a node creation followed by an attachment to existing nodes using a GNC-Attach. To us, this represents the creation of a new class implementing a new feature. This new feature depends upon existing classes. The point of being attached to all dependent classes of a class means that those classes are already used, depending upon each others. If class X depends on classes A, B and C, it means that A, B and C interact together in a way that is defined by X. When a new node n_i is connected to X by way of a GNC-Attach, it is also connected to A, B and C. In other words, *the new class n_i creates a novel interaction between A, B, and C.*

Executing GNC-Attach twice may reflect the fact that the new class mixes two existing groups of already interacting classes. By group of already interacting classes, we mean that each class of the group uses or is used (method call, field access) by another class in the group. In the model, there is never more than two groups of already interacting classes being linked from a new node (a new feature). We did experiments allowing more than 2 successive GNC-Attach in the GD-GNC main loop: this has never significantly increased the fit to real data. *One possible explanation to this observation might be that mixing meaningfully and correctly two groups of already interacting classes is already quite a hard operation, and it happens very rarely to mix more than two such groups.*

The second basic operation of GD-GNC (the top-level else condition) is a reverse attachment from an existing node to the added node. For us, it may be explained as a refactoring operation, where a piece of code is extracted from an existing class, in order to ease reuse and to simplify the code. Once the refactoring is performed, the newly created class is ready for being reused. This is what can happen in subsequent iterations of the algorithm with the GNC-Attach.

To us, these are the most likely explanations to why the GD-GNC model we propose fits real software graphs. It is worth noting that we have performed experiments with many other different models not reporting here. They embedded mechanisms corresponding to various assumptions on what a new feature or a refactoring may be (according to the common software engineering sense or our own programming experience). They all led to a poor fit in terms of degree distribution.

To sum up, the two core operations GD-GNC can be explained as: (i) representing the creation of a new feature by remix; (ii) refactoring.

5.2 Threats to Validity

Let us now discuss the threats to the validity of our findings. First, we have optimized our model with respect to the fit to in-degree and out-degree distributions. Even

TABLE 3: δ minimum, median and maximal values for GD-GNC, Baxter & Freen and Erdős & Rényi models for 30 generations of each model (*cf.* section 4.3 for more information on δ value). The last column is the p-value determined using the Mann-Whitney test; this p-value assesses whether all values of GD-GNC are significantly always smaller than Baxter & Freen’s ones.

Software	Erdős & Rényi			Baxter & Freen			GD-GNC			p-value
	Min	Med	Max	Min	Med	Max	Min	Med	Max	
ant	0.77	0.91	0.97	0.32	0.51	0.82	0.29	0.45	0.63	$\leq 10^{-3}$
jfreechart	0.66	0.81	0.91	0.17	0.39	0.58	0.13	0.20	0.39	$\leq 10^{-8}$
jftp	0.52	0.70	0.82	0.22	0.44	0.82	0.08	0.27	0.58	$\leq 10^{-6}$
jtds	0.47	0.64	0.81	0.24	0.40	0.72	0.17	0.31	0.37	$\leq 10^{-5}$
maven	0.63	0.76	0.81	0.09	0.22	0.46	0.24	0.46	0.51	$\leq 10^{-10}$
hsqldb	0.58	0.71	0.83	0.14	0.30	0.46	0.51	0.62	0.65	$\leq 10^{-11}$
log4j	0.69	0.83	0.92	0.15	0.41	0.72	0.23	0.40	0.50	0.381
squirreelsql	0.65	0.85	0.94	0.26	0.47	0.78	0.19	0.29	0.43	$\leq 10^{-9}$
argouml	0.80	0.92	0.97	0.27	0.56	0.73	0.20	0.38	0.61	$\leq 10^{-4}$
mvnforum	0.57	0.69	0.78	0.12	0.30	0.53	0.19	0.39	0.45	$\leq 10^{-3}$
atunes	0.64	0.83	0.94	0.17	0.40	0.77	0.23	0.38	0.47	0.285
jedit	0.74	0.87	0.95	0.23	0.52	0.77	0.11	0.21	0.33	$\leq 10^{-11}$
weka	0.74	0.89	0.97	0.24	0.49	0.80	0.15	0.42	0.56	$\leq 10^{-5}$
jetty	0.70	0.87	0.94	0.12	0.48	0.72	0.23	0.41	0.55	$\leq 10^{-3}$
vuze	0.62	0.75	0.88	0.12	0.26	0.43	0.22	0.31	0.37	$\leq 10^{-3}$

if the degree distributions capture many topological properties of graphs, it is only one facet of the structure of the dependency graph. One threat to the validity of our conclusions is that some other important topological properties of software dependency graphs have minor or no impact to degree distributions.

Second, our experiments are done on a dataset of 15 Java software systems. Our findings may only hold for object-oriented code, Java software or even worse, to this particular dataset only. However, for us, a sign of hope is that the degree distributions on other programming languages and systems that are reported in previous work look qualitatively the same [5], [6], [12].

Third, our evolution model is completely expressed in abstract graph terms. We have reformulated the algorithm in a software engineering perspective in section 5.1. It may be the case that we have correctly extracted the core operations but that, at the same time, we have mis-interpreted their meaning. We look forward to more work in this area, to discuss with the community in order to see the emergence of a consensus on the core software evolution mechanisms.

6 RELATED WORK

Several authors have proposed models for generating directed graphs in various domains. Kumar et al. [19] and Bollobás et al. [20] have proposed models intended to generate graphs looking like the World Wide Web graph. Grindrod [21] has proposed a model related to protein identification on bioinformatics. Many other models are generic: Erdős & Rényi [18] one, Dorogovtsev et al. [22] one or Vazquez [23] one are example of generic models. The R-MAT model proposed by Chakrabarti et al. [24]

is also generic, but its generation process is based on matrix operations. According to our experiments, those models are not able to reproduce software dependency graphs cumulative degree distribution.

Baxter *et al.* [2] studied a large amount of metrics, including graph metrics; Louridas *et al.* [5] studied the “pervasive” presence of power-law distributions on software dependencies graphs at the class and features level for a large range of software written in various languages. Myers [12] also studied graph metrics on software. Nevertheless, none of them showed the common structure across software degree distributions as we have presented in this paper. Our results on the asymmetry between the in-degree and out-degree distributions confirm previous findings by Meyers [12], Challet and Lombardoni [15], Valverde and Solé [6] and Baxter and Freen [2].

The tendency of software graphs to follow a growth mechanism similar to the GNC-Attach one has been reported by Valverde and Solé [6]. Moreover, they used a model of duplication and rewiring to generate software graphs with similar motifs [17]. Similarly, Myers [12] proposed a generative model based on binary strings to materialize the software evolution rules. Unfortunately, our investigation shown us that both models has bad degree distribution fits with real dependency software graphs. Baxter and Freen proposed a generative model of software graphs [2], based on a preferential attachment which depends on the node degree distributions. We have shown that our model better fits real data.

Maddison and Tarlow [25] proposed a generative model of source code. Our motivations are similar but the considered software artifacts are completely different

(abstract syntax tree versus dependency graphs).

Some authors have studied other structure characteristics on different kinds of graphs: Harman *et al.* [26] focus on dependency clusters to demonstrate the widespread existence of clusters in software source code. They show a common traits of software using a different approach. Mitchell and Mancoridis [27] uses clustering techniques to infer aggregate view of a software system but they want to gain understanding for specific systems in order to improve debugging and refactoring but they do not focus on generalities about software. Furthermore, none of these studies propose generative model of any kind. Chaikalis and Chatzigeorgiou [28] proposed a model able to

7 CONCLUSION

In this paper, we have studied the fact that there is a common structure in many software dependency graphs, and devised an experimental protocol to understand the evolution principles that result in such a common structure. Our experimental approach is to devise a generative model of software dependency graphs and to compare the structure of artificial graphs generated with model against real software graphs.

We have introduce a new generative model GD-GNC. This model generates graphs whose degree distribution seem very close to that of real software. This closeness is assessed with statistical tests. The basic operations which make up the model tell us something about the way software evolves: according to our experiments, new features are based on the perpetual remix of existing interacting classes and refactoring mostly consists of extracting a reusable class from an existing class.

Now that we have shown that meaningful generative models exist, future work can go beyond degree distributions. Graph motifs are patterns consisting of a small amount of nodes connected to each other in a certain way. Graph motifs [29] may turn to be valuable to determine the structure closeness of generated graphs. We hypothesize that motifs also emerge from the core evolution rules. Hence, if generated graphs share similar motifs with real software graphs, there is a good chance that the core evolution primitives of the model are close to the real ones.

REFERENCES

- [1] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a Case Study: its Extracted Software Architecture," in *Proceedings of the 21st International Conference on Software Engineering*. New York, NY, USA: ACM, 1999, pp. 555–563.
- [2] G. J. Baxter and M. R. Frean, "Software Graphs and Programmer Awareness," *ArXiv e-prints*, Feb 2008.
- [3] P. Bhattacharya, M. Iliofotou, I. Neamtii, and M. Faloutsos, "Graph-based Analysis and Prediction for Software Evolution," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 419–429.
- [4] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "On the Suitability of yule Process to Stochastically Model some Properties of Object-Oriented Systems," *Physica A: Statistical Mechanics and its Applications*, vol. 370, no. 2, pp. 817 – 831, 2006.
- [5] P. Louridas, D. Spinellis, and V. Vlachos, "Power Laws in Software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 2:1–2:26, Oct 2008.
- [6] S. Valverde and R. V. Solé, "Logarithmic Growth Dynamics in Software Networks," *EPL (Europhysics Letters)*, vol. 72, no. 5, p. 858, 2005.
- [7] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, "The Web as a Graph: Measurements, Models, and Methods," in *Computing and Combinatorics*, T. Asano, H. Imai, D. Lee, S. ichi Nakano, and T. Tokuyama, Eds. Springer Berlin Heidelberg, 1999, pp. 1–17.
- [8] L. Li, D. Alderson, J. C. Doyle, and W. Willinger, "Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications," *Internet Mathematics*, vol. 2, no. 4, pp. 431–523, 2005.
- [9] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.
- [10] M. Newman, *Networks: An Introduction*. Oxford University Press, 2010.
- [11] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free Geometry in OO Programs," *Communications of the ACM*, vol. 48, no. 5, pp. 99–103, May 2005.
- [12] C. R. Myers, "Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs," *Physical Review E*, vol. 68, p. 046116, Oct 2003.
- [13] P. L. Krapivsky and S. Redner, "Network Growth by Copying," *Physical Review E*, vol. 71, p. 036118, Mar 2005.
- [14] J. D. Gibbons and S. Chakraborti, *Nonparametric Statistical Inference, Fourth Edition: Revised and Expanded*. CRC Press, Mar 2014.
- [15] D. Challet and A. Lombardoni, "Bug Propagation and Debugging in Asymmetric Software Structures," *Physical Review E*, vol. 70, p. 046109, Oct 2004.
- [16] E. F. Keller, "Revisiting "Scale-free" Networks," *BioEssays*, vol. 27, no. 10, pp. 1060–1068, 2005.
- [17] S. Valverde and R. V. Solé, "Network Motifs in Computational Graphs: A Case Study in Software Architecture," *Physical Review E*, vol. 72, p. 026107, Aug 2005.
- [18] P. Erdős and A. Rényi, "On Random Graphs," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297.
- [19] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Stochastic Models for the Web Graph," in *41st Annual Symposium on Foundations of Computer Science, 2000. Proceedings.*, 2000, pp. 57–65.
- [20] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan, "Directed Scale-free Graphs," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 132–139.
- [21] P. Grindrod, "Range-dependent Random Graphs and Their Application to Modeling Large Small-world Proteome Datasets," *Physical Review E*, vol. 66, p. 066702, Dec 2002.
- [22] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin, "Structure of Growing Networks with Preferential Linking," *Physical Review Letters*, vol. 85, pp. 4633–4636, Nov 2000.
- [23] A. Vazquez, "Knowing a Network by Walking on it: Emergence of Scaling," *ArXiv e-prints*, Jun 2000.
- [24] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 2004, pp. 442–446.
- [25] C. Maddison and D. Tarlow, "Structured Generative Models of Natural Source Code," *ArXiv e-prints*, Jan 2014.
- [26] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence Clusters in Source Code," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 1, pp. 1:1–1:33, Nov 2009.
- [27] B. S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, Mar 2006.
- [28] T. Chaikalis and A. Chatzigeorgiou, "Forecasting Java Software Evolution Trends employing Network Models," *IEEE Transactions on Software Engineering*, 2015.
- [29] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.